REMARKS

Claims 9 and 11 have been previously canceled. Claims 1, 12, and 21 have been amended. Claims 1 through 8, 10, and 12 through 21 remain in the application.

35 U.S.C. § 102

Claims 1 through 8, 10, and 12 through 21 were rejected under 35 U.S.C. § 102(b) as being anticipated by "Emulation of a Material Delivery System", by Todd LeBaron and Kelly Thompson. Applicants respectfully traverse this rejection.

The "Emulation of a Material Delivery System", by Todd LeBaron and Kelly Thompson, discloses emulation of a complex pick and pack system. A material handling system consists of conveyor sections which continuously move carriers around a closed loop that connects all pick and pack stations. Routing logic, PLC or PC control software, sequencing algorithms and more can be integrated, tested, and debugged within a simulation environment. Emulation has been used for a Rapistan Systems Project to test, debug, and optimize complex algorithms and control logic. Emulation of the Rapistan control system for this project integrates a simulation model with the actual control system. The simulation model provides the output for evaluating control logic and algorithms. The emulation used at Rapistan Systems was able to prove that the system could handle the projected growth in daily orders. Emulation provides the graphical and statistical output needed to accurately evaluate different algorithms and control logic. LeBaron et al. does not disclose playing a simulation model by a PLC logical verification system on a computer and allowing a user to visually see flow of a part through the manufacturing line, wherein the PLC logical verification system dynamically interacts through input and output with the simulation model to verify a PLC code of the manufacturing line, and generating the PLC code if a part flow represented in the simulation model is correct. LeBaron

et al. also does <u>not</u> disclose using the generated PLC code and implementing the manufacturing line according to the part flow simulation model.

In contradistinction, independent claim 1, as amended, clarifies the invention claimed as a method of part flow for a programmable logic controller logical verification system. The method includes the steps of constructing a simulation model of a manufacturing line using a computer, playing the simulation model by a PLC logical verification system on the computer and allowing a user to visually see flow of a part through the manufacturing line. The PLC logical verification system dynamically interacts through input and output with the simulation model to verify a PLC code of the manufacturing line. The method also includes the steps of determining if the part flow represented in the simulation model is correct and generating PLC code if the part flow represented in the simulation model is correct. The method further includes the steps of using the generated PLC code and implementing the manufacturing line according to the part flow simulation model. Independent claims 12 and 21 have been amended similar to claim 1 and include other features of the present invention.

A rejection grounded on anticipation under 35 U.S.C. § 102 is proper only where the subject matter claimed is identically disclosed or described in a reference. In other words, anticipation requires the presence of a single prior art reference which discloses each and every element of the claimed invention arranged as in the claim. <u>In re Arkley</u>, 455 F.2d 586, 172 U.S.P.Q. 524 (C.C.P.A. 1972); <u>Kalman v. Kimberly-Clark Corp.</u>, 713 F.2d 760, 218 U.S.P.Q. 781 (Fed. Cir. 1983); <u>Lindemann Maschinenfabrik GMBH v. American Hoist & Derrick Co.</u>, 730 F.2d 1452, 221 U.S.P.Q. 481 (Fed. Cir. 1984).

LeBaron et al. does <u>not</u> disclose or anticipate the claimed invention of claims 1 through 8, 10, and 12 through 21. Specifically, LeBaron et al. <u>merely</u> discloses an emulation of a material delivery system in which routing logic, PLC or PC control software, sequencing

algorithms and more can be integrated, tested, and debugged within a simulation environment. LeBaron et al. lacks playing a simulation model by a PLC logical verification system on a computer and allowing a user to visually see flow of a part through the manufacturing line wherein the PLC logical verification system dynamically interacts through input and output with the simulation model to verify a PLC code of the manufacturing line. In LeBaron et al., there is no PLC logical verification system and no PLC code is generated. As is known in the art, an emulator represents a physical device in software. (See Wikipedia dictionary definition from Wikipedia website, copy attached). The PLC logical verification system is not a software representation of a physical device, but a software tool that allows dynamic interaction directly with a simulation model to test PLC logic by having an input and output exchange similar to input/output control logic to validate that the logic is delivering what is intended. The PLC logical verification system analytically verifies the PLC logic. (A similar system is disclosed in U.S. Patent No. 6,442,441). In LeBaron et al., the emulator does not validate that the logic is delivering what is intended.
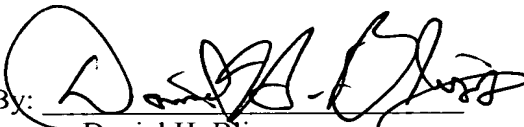
LeBaron et al. also lacks using the generated PLC code and implementing the manufacturing line according to the part flow simulation model. In LeBaron et al., while LeBaron et al. mentions PLC or PC control software can be tested and debugged within a simulation environment, there is no part flow for a programmable logic controller logical verification system and there is no generated PLC code that is used in implementing a manufacturing line.

As such, LeBaron et al. fails to disclose the combination of a method of part flow for a programmable logic controller logical verification system including the steps of constructing a simulation model of a part flow in a manufacturing line using a computer, playing the simulation model by a PLC logical verification system on the computer and allowing a user

to visually see flow of a part through the manufacturing line, wherein the PLC logical verification system dynamically interacts through input and output with the simulation model to verify a PLC code of the manufacturing line, determining if the part flow represented in the simulation model is correct, generating PLC code if the part flow represented in the simulation model is correct, using the generated PLC code, and implementing the manufacturing line according to the part flow simulation model as claimed by Applicants. Therefore, it is respectfully submitted that claims 1 through 8, 10, and 12 through 21 are allowable over the rejection under 35 U.S.C. § 102(b).

Based on the above, it is respectfully submitted that the claims are in a condition for allowance or in better form for appeal. Applicants respectfully request reconsideration of the claims and withdrawal of the final rejection. It is respectfully requested that this Amendment be entered under 37 C.F.R. 1.116.

Respectfully submitted,

By: _____
Daniel H. Bliss
Registration No. 32,398

BLISS McGLYNN, P.C.
2075 West Big Beaver, Suite 600
Troy, Michigan 48084
(248) 649-6090

Date: February 5, 2007

Attorney Docket No.: 0693.00242
Ford Disclosure No.: 200-0664

Emulator

From Wikipedia, the free encyclopedia

A **software emulator** allows computer programs to run on a platform (computer architecture and/or operating system) other than the one for which they were originally written. Unlike simulation, which attempts to gather a great deal of runtime information as well as reproducing a program's behavior, emulation attempts to model to various degrees the state of the device being emulated. High-level emulation uses a combination of the two approaches in an attempt to retain as much accuracy as possible while having the advantages of simplicity and speed provided by simulation.

A **hardware emulator** is an emulator which takes the form of a hardware device. Examples include printer emulators inside the ROM of the printer, and FPGA-based emulators.

An emulator reproducing Final Fantasy III's environment on a Windows computer.

In a theoretical sense, the Church-Turing thesis implies that any operating environment can be emulated within any other. In practice, it can be quite difficult, particularly when the exact behavior of the system to be emulated is not documented and has to be deduced through reverse engineering. It also says nothing about timing constraints; if the emulator does not perform as quickly as the original hardware, the emulated software may run much more slowly than it would have on the original hardware.
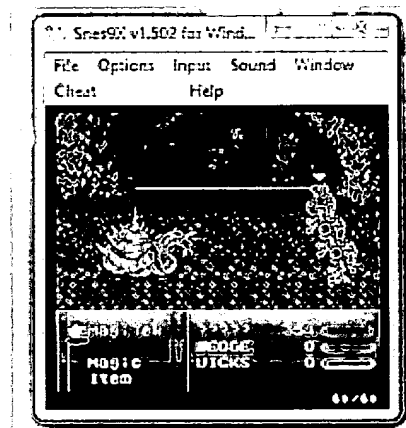
Emulation refers to the ability of a program or device to **imitate** another program or device. Many printers, for example, are designed to emulate Hewlett-Packard LaserJet printers because so much software is written for HP printers. By emulating an HP printer, a printer can work with any software written for a real HP printer. Emulation "tricks" the software into believing that a device is really some other device.

Contents

Structure

Most emulators just emulate a hardware architecture — if operating system firmware or software is required for the desired software, it must be provided as well (and may itself be emulated). Both the OS and the software will then be interpreted by the emulator, rather than being run by native hardware. Apart from this interpreter for the emulated machine's language, some other hardware (such as input or

output devices) must be provided in virtual form as well; if writing to a specific memory location should influence the screen, for example, this will have to be emulated.

While emulation could, if taken to the extreme, go down to the atomic level, basing its output on a simulation of the actual circuitry from a virtual power source, this would be a highly unusual solution. Emulators typically stop at a simulation of the documented hardware specifications and digital logic. Sufficient emulation of some hardware platforms requires extreme accuracy, down to the level of individual clock cycles, undocumented features, unpredictable analog elements, and implementation bugs. This is particularly the case with classic home computers such as the Commodore 64, whose software often depends on highly sophisticated low-level programming tricks invented by game programmers and the demoscene.

In contrast, some other platforms have had very little use of direct hardware addressing. In these cases, a simple compatibility layer may suffice. This translates system calls for the emulated system into system calls for the host system.

Developers of software for embedded systems or video game consoles often design their software on especially accurate emulators called simulators before trying it on the real hardware. This is so that software can be produced and tested before the final hardware exists in large quantities, so that it can be tested without taking the time to copy the program to the hardware, or so that it can be debugged at a low level without introducing the side effects of a debugger. In many cases, the simulator is actually produced by the company providing the hardware, which theoretically increases its accuracy.

Typically, an emulator is divided into modules that correspond roughly to the emulated computer's subsystems. Most often, an emulator will be composed of the following modules:

- a CPU emulator or CPU simulator (the two terms are mostly interchangeable in this case)
- a memory subsystem module
- various I/O devices emulators

Buses are often not emulated, either for reasons of performance or simplicity, and virtual peripherals communicate directly with the CPU or the memory subsystem.

A detailed description of the internals of a specific emulator can be found in the ElectrEm article.

Memory subsystem

It is possible for the memory subsystem emulation to be reduced to simply an array of elements each sized like an emulated word; however, this model falls very quickly as soon as any location in the computer's logical memory does not match physical memory.

This clearly is the case whenever the emulated hardware allows for advanced memory management (in which case, the MMU logic can be embedded in the memory emulator, made a module of its own, or sometimes integrated into the CPU simulator).

Even if the emulated computer does not feature an MMU, though, there are usually other factors that break the equivalence between logical and physical memory: many (if not most) architecture offer memory-mapped I/O; even those that do not almost invariably have a block of logical memory mapped to ROM, which means that the memory-array module must be discarded if the read-only nature of ROM

is to be emulated. Features such as bank switching or segmentation may also complicate memory emulation.

As a result, most emulators implement at least two procedures for writing to and reading from logical memory, and it is these procedures' duty to map every access to the correct location of the correct object.

On a base-limit addressing system where memory from address *0* to address *ROMSIZE* is read-only memory, while the rest is RAM, something along the line of the following procedures would be typical:

```
void WriteMemory(word Address, word Value) {
    word RealAddress;
    RealAddress=Address+BaseRegister;
    if(RealAddress<LimitRegister) {
        if(RealAddress>ROMSIZE) Memory[RealAddress]=Value;
    } else {
        RaiseInterrupt(INT_SEGFAULT);
    }
}
```

```
void ReadMemory(word Address) {
    word RealAddress;
    RealAddress=Address+BaseRegister;
    if(RealAddress<LimitRegister) {
        return Memory[RealAddress];
    } else {
        RaiseInterrupt(INT_SEGFAULT);
        return NULL;
    }
}
```

CPU simulator

The CPU simulator is often the most complicated part of an emulator. Many emulators are written using "pre-packaged" CPU simulators, in order to concentrate on good and efficient emulation of a specific machine.

The simplest form of a CPU simulator is an interpreter, which follows the execution flow of the emulated program code and, for every machine code instruction encountered, executes operations on the host processor that are semantically equivalent to the original instructions.

This is made possible by assigning a variable to each register and flag of the simulated CPU. The logic of the simulated CPU can then more or less be directly translated into software algorithms, creating a software re-implementation that basically mirrors the original hardware implementation.

The following example illustrates how CPU simulation is accomplished by an interpreter. In this case, interrupts are checked-for before every instruction executed, though this behavior is rare in real emulators for performance reasons.

```
void Execute(void) {
    if(Interrupt!=INT_NONE) {
        SuperUser=TRUE;
        WriteMemory(++StackPointer, ProgramCounter);
        ProgramCounter=InterruptPointer;
    }
    switch(ReadMemory(ProgramCounter++)) {
```

```
        /*
         * Handling of every valid instruction
         * goes here...
         */
        default:
        Interrupt=INT_ILLEGAL;
    }
}
```

Interpreters are very popular as computer simulators, as they are much simpler to implement than more time-efficient alternative solutions, and their speed is more than adequate for emulating computers of more than roughly a decade ago on modern machines.

However, the speed penalty inherent in interpretation can be a problem when emulating computers whose processor speed is on the same order of magnitude as the host machine. Until not many years ago, emulation in such situations was considered completely impractical by many.

What allowed breaking through this restriction were the advances in dynamic recompilation techniques. Simple *a priori* translation of emulated program code into code runnable on the host architecture is usually impossible because of several reasons:

- code may be modified while in RAM, even if it is modified only by the emulated operating system when loading the code (for example from disk)
- there may not be a way to reliably distinguish data (which should not be translated) from executable code.

Various forms of dynamic recompilation, including the popular Just In Time compiler (JIT) technique, try to circumvent these problems by waiting until the processor control flow jumps into a location containing untranslated code, and only then ("just in time") translates a block of the code into host code that can be executed. The translated code is kept in a *code cache*, and the original code is not lost or affected; this way, even data segments can be (meaninglessly) translated by the recompiler, resulting in no more than a waste of translation time.

I/O

Most emulators do not, as mentioned earlier, emulate the main system bus; each I/O device is thus often treated as a special case, and no consistent interface for virtual peripherals is provided.

This can result in a performance advantage, since each I/O module can be tailored to the characteristics of the emulated device; designs based on a standard, unified I/O API can however rival such simpler models, if well thought-out, and they have the additional advantage of "automatically" providing a plug-in service through which third-party virtual devices can be used within the emulator.

A unified I/O API may not necessarily mirror the structure of the real hardware bus: bus design is limited by several electric constraints and a need for hardware concurrency management that can mostly be ignored in a software implementation.

Even in emulators that treat each device as a special case, there is usually a common basic infrastructure for:

- managing interrupts, by means of a procedure that sets flags readable by the CPU simulator

whenever an interrupt is raised, allowing the virtual CPU to "poll for (virtual) interrupts"
- writing to and reading from physical memory, by means of two procedures similar to the ones dealing with logical memory (although, contrary to the latter, the former *can* often be left out, and direct references to the memory array be employed instead)

Legal Controversy

See article Console emulator - Legal Issues

Nintendo[1] (http://www.nintendo.com/corp/legal.jsp)
Other[2] (http://www.cnn.com/TECH/computing/9806/02/game.emulators.idg/index.html)

See also

- List of emulators - Software emulators:
  - Arcade emulator
  - Console emulator
- Other uses of the term emulator in the field of computer science:
  - Server emulator
  - Terminal emulator
  - Game engine recreations
- Hardware emulators:
  - Field Programmable Gate Array (FPGA) Arcade hardware emulators.
  - Field Programmable Gate Array (FPGA) CPU hardware emulators.
- Translation:

Binary translation

- In-circuit emulator (ICE)
- Virtual machine
- Virtual Console
- Source port

External links

- Neogeone (http://www.neogeone.com/), emulators / roms / Neo-Geo / CPS1 / CPS2
- *HowTo: Writing a Computer Emulator (http://fms.komkon.org/EMUL8/HOWTO.html)*
- WABI - Application Programming Interface (API) Translator from Sun Microsystem (http://www.sun.com/smi/Press/sunflash/1996-05/sunflash.960528.10612.xml)
- Checklist (http://checklist.berzerk.co.uk/) - Emulator Compatibility lists
- *The History of Emulation - 1800 to 1999:* Part 1 (http://www.zophar.net/articles/art_14-1.html), Part 2 (http://www.zophar.net/articles/art_14-2.html), Part 3 (http://www.zophar.net/articles/art_14-3.html), Part 4 (http://www.zophar.net/articles/art_14-4.html)
- EmuFAQ (http://www.overclocked.org/emufaq/EmuFAQ_index.htm) - collection of essays surrounding the legality of emulators
- A Simple Python CPU Emulator Implementation (http://www.evilbitz.com/2006/12/23/a-simple-python-cpu-emulator/)
- Emulator News - Free Online Games - and Emulator downloads (http://www.vg-network.com/)

Retrieved from "http://en.wikipedia.org/wiki/Emulator"

Category: Emulation software